Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий

Кафедра «Информационные системы»

УТВЕРЖДАЮ
Заведующий кафедрой ИС
_____ С. А. Виденин
«___» _____ 2016 г.

БАКАЛАВРСКАЯ РАБОТА

09.03.02 Информационные системы и технологии

Разработка ИС для центра дополнительного и безотрывного образования по ИТ ИКИТ

Руководитель _____ С.А. Виденин

Выпускник _____ Э.В. Васильев

Нормоконтролер _____ Ю. В. Шмагрис

Красноярск 2016

СОДЕРЖАНИЕ

Введение 3
1 Теоретическая часть 5
1.1 Предметная область 5
1.2 Возможные решения 6
1.3 Рассматриваемые подходы 8
1.3.1 Сервис-ориентированная архитектура 8
1.3.2 Композитная архитектура 13
1.4 Вывод 26
2 Практическая часть 27
2.1 Инструментальные средства разработки 27
2.2 Архитектура и процесс разработки 30
2.2.1 Технология ASP.NET 31
2.2.2 ORM Entity Framework 33
2.2.3 IoC-контейнер Autofac 37
2.3 Структура проекта 38
2.4 Реализация 41
Заключение 46
Список использованных источников 47

ВВЕДЕНИЕ

Современные технологии предоставляют рынку широкий спектр возможностей для решения различного вида задач. И сфера обучения не является исключением. Повышение доступности знаний, ускорение процесса обучения и получения быстрой отдачи при одновременном сокращении издержек являются сейчас актуальными потребностями для многих коммерческих организаций.

Для образовательных организаций актуально наличие информационной системы, которая объединит в себе образовательные ресурсы, средства обучения и управления образовательным процессом.

Центр дополнительного и безотрывного образования по информационным технологиям основан на базе Института космических и информационных технологий. Направлен на повышения квалификации студентов, сотрудников университета, а также слушателей внешней аудитории. Центр включает в себя 9 программ дополнительного образования:

— построение корпоративных вычислительных сетей на базе технологии Cisco Systems;

— информационная безопасность в корпоративных сетях;

— создание и администрирование телекоммуникационного узла на базе ОС Linux;

— специалист в области компьютерной графики и web-дизайна;

— компьютерная графика и верстка;

— переводчик в сфере профессиональной коммуникации;

— английский для начинающих;

— разговорный английский язык;

— разработчик программного обеспечения (.NET разработчик).

Более подробную информацию можно получить на сайте института [2].

Исходя из этого была поставлена цель: повысить эффективность бизнес-процессов центра дополнительного и безотрывного образования по ИТ ИКИТ.

Для достижения поставленной цели были определены следующие задачи:

— изучить предметную область;

— рассмотреть существующие аналоги систем автоматизации процессов обучения;

— проанализировать временные затраты внедрения информационной системы;

— разработать приложение для центра дополнительного образования.

1 Теоретическая часть

1.1 Предметная область

Проведя несколько бесед с заказчиком, была изучена и проанализирована предметная область. Чтобы избежать несоответствий с и удостовериться, что предметная область понята правильно, была построена диаграмма use-case. Она представлена на рисунке 1.1.1.

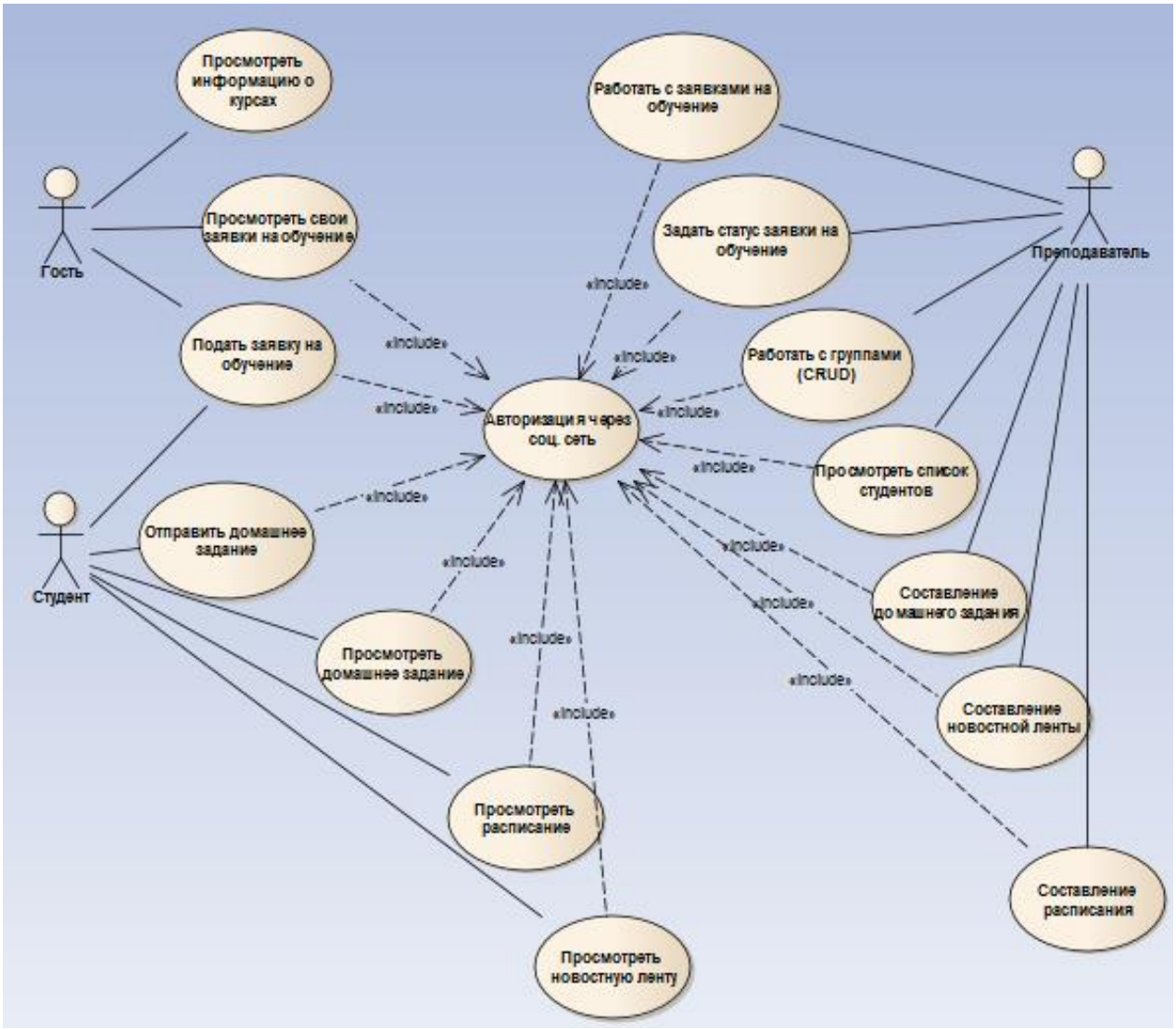

Рисунок 1.1.1 – Use-case диаграмма

Было принято решение повысить эффективность бизнес-процессов центра дополнительного образования путем внедрения информационной системы. Однако мы выявили несколько потенциальных проблем:

— бизнес-процессы не до конца поставлены;

— бизнес-процессы будут меняться в ближайшем будущем.

Следовательно, необходимо выбрать информационную систему, которая позволит гибко и безболезненно вносить в неё изменения.

1.2 Возможные решения

Мы решили изучить уже готовые аналоги подобных систем. Было рассмотрено две системы:

— 1С: Электронное обучение;

— Moodle.

«1С: Электронное обучение» представляет собой линейку программных продуктов, предназначенных для электронного и смешанного обучения в различных организациях. Система обеспечивает индивидуальный, а также многопользовательский доступ через сеть Интернет. Важным достоинством является адаптивность системы. Курсы можно обновлять и дополнять своими материалами, после чего пользователь получит уведомление об изменении курса.

Однако, «1С: Электронное обучение» имеет ряд недостатков.

Во-первых, высокая цена системы. Приобретение всех продуктов линейки обходится в большую сумму денег.

Во-вторых, «1С: Электронное обучение» является desktop-приложением. Это усложняет процесс внедрения и сопровождения, т.к. каждый студент и работник должен будет установить систему на свой компьютер. Из этого следует, что нам необходима система, реализованная как web-приложение. Это обеспечит возможность входа в неё из любого устройства, имеющего доступ в сеть Интернет.

В-третьих, данная система не покрывает все бизнес-процессы, которые нам необходимо автоматизировать.

Moodle – это модульная объектно-ориентированная динамическая учебная среда; пакет, который обычно определяют как CMS или LMS. Эти аббревиатуры можно расшифровать следующим образом:

— CMS (course managment system) - система управления курсами;

— LMS (learning managment system) - система управления обучением.

Moodle реализована на языке программирования PHP. Популярность ей обеспечил открытый исходный код. Сама система распространяется по лицензии GNU GPL, это означает, что не придется производить лицензионные отчисления в пользу разработчика.

Основной учебной единицей Moodle являются учебные курсы. В рамках такого курса можно организовать:

— взаимодействие учеников между собой и с учителем. Для этого используются форумы, чаты;

— передачу знаний в электронном виде с помощью файлов, архивов, веб-страниц, лекций;

— проверку знаний и обучение с помощью тестов и заданий. Результаты работы отправляются в текстовом виде или в виде файлов.

Учитывая плюсы, которые предоставляет данное решение, оно также имеет ряд недостатков.

Во-первых, одной из первых проблем, с которой могут столкнуться желающие организовать электронное обучение, является решение технических вопросов связанных с этой системой. Это объясняется, в первую очередь, отсутствием доступных и грамотно составленных инструкций и рекомендации по работе с системой на русском языке, то есть возникает проблема с поддержкой данной системы.

Во-вторых, для решения наших задач обязательна автоматизация приема заявок и работы с ними, чего нет в Moodle. Возникает необходимость написания дополнительных модулей в системе, в которой, возможно, есть ряд неочевидных ограничений.

В-третьих, бизнес-процессы неустоявшиеся, предположить, что будет в конце разработки и куда уйдет бизнес невозможно, а данная система рассчитана только на работу с курсами и систему управления обучением.

Рассмотрев и проанализировав аналогичные системы, мы пришли к выводу, что лучшим решением поставленной цели станет разработка собственной системы для центра дополнительного и безотрывного образования по ИТ ИКИТ, которая покроет все бизнес-процессы, присущие данной предметной области. А также, предусмотрит возможность обновления и расширения системы новыми модулями.

1.3 Рассматриваемые подходы

Перед нами встала задача: разработать ИС для центра дополнительного и безотрывного образования по ИТ ИКИТ. Однако, так как бизнес-процессы не устоялись, необходимо определиться с архитектурой, которая даст возможность внедрять изменения в систему по ходу изменения или добавления новых бизнес-процессов.

1.3.1 Сервис-ориентированная архитектура

Сервис-ориентированная архитектура (англ. service-oriented architecture, SOA) – модульный подход к разработке программного обеспечения, основанный на использовании распределённых, слабо связанных (англ. loose coupling) взаимозаменяемых компонентов (сервисов), оснащённых стандартизированными интерфейсами для взаимодействия по стандартизированным протоколам. SOA – это не технология, а способ проектирования и организации информационной архитектуры приложения, а также его бизнес-функциональности.

Как подход к организации распределенной вычислительной инфраструктуры, задача SOA заключается в построении данной

инфраструктуры, отталкиваясь от решаемых задач, а не используемых при этом технологий. Акцент делается на совместном и повторном использовании типовой функциональности, оформленной в виде доступных по сети сервисов. Новые приложения могут создаваться путем обнаружения и композиции существующих сервисов. Функциональность сервиса может одновременно использоваться в контексте сразу нескольких приложений. При этом сервису может быть не известно о том, в контексте каких приложений он используется. Кроме того, при выполнении запросов сервис может использовать функциональность других сервисов. Таким образом, в SOA достигается переход от монолитных распределенных приложений к приложениям, состоящим из набора слабо связанных распределенных компонентов, обнаруживаемых динамически в сети. [3]

Главное отличие SOA в том, что она разрешает одновременное использование и обмен данными между программными системами от различных поставщиков без необходимости вносить изменения в исходный код самих систем или сервисов.

Таким образом, системы, основанные на SOA, могут быть независимы от технологий разработки и платформ (таких как Java, .NET и т.д.). К примеру, сервисы, написанные на C#, работающие на платформах .Net и сервисы на Java, работающие на платформах Java EE, могут быть с одинаковым успехом вызваны общим составным приложением. Приложения, работающие на одних платформах, могут вызывать сервисы, работающие на других платформах, что облегчает повторное использование компонентов.

SOA находит применение в информационных системах уровня предприятия из самых различных областей: коммерция, развлекательные сервисы, обучение, менеджмент, здравоохранение и т.д.

Особенности SOA:

— повторное использование сервисов (акцент на использовании уже имеющихся сервисов, а не повторной реализации новых служб);

— абстракция (сервисы не зависят от реализации и платформы);

— взаимодействие по контрактам (сервисы предоставляют интерфейсы взаимодействия, не зависящие от реализации приложений их использующих);

— формальное описание (сервисы накладывают определенные ограничения между поставщиками и потребителями сервисов);

— релевантность (сервисы реализуют конкретные бизнес-процессы, которыми оперируют конечные пользователи).

Следует отметить, что SOA – это больше чем просто архитектура. Кроме архитектурных аспектов, для внедрения полноценного SOA-решения необходимы также механизмы мониторинга и управления сервисами, детально разработанный уровень инфраструктуры (как правило им служит сервисная шина), а также уровень архитектуры данных.

На рисунке 1.3.1 показаны основные компоненты типичной сервис-ориентированной архитектуры.

Рисунок 1.3.1 – Компоненты сервис-ориентированной архитектуры

Приложения: предоставляют бизнес-функционал сервисов конечным пользователям.

Сервисы: реализуют бизнес-логику на верхнем уровне.

Контракты: определяют назначение и функциональные ограничения сервисов.

Интерфейсы: определяют функциональность сервисов.

Реализация: обеспечивает выполнение бизнес-логики и работу с данными.

Хранилище сервисов: регистрирует сервисы и их параметры, определяет права доступа к сервисам.

Сервисная шина: обеспечивает взаимодействие между сервисами и приложениями.

Важно понимать, что SOA – это не способ проектирования новых систем, а подход к улучшению и сопровождению уже существующих крупных программных систем [3]. Внедрение SOA дает следующие преимущества:

— синхронизация бизнес-процессов и технологий для достижения быстрой реакции в ответ на изменения в предметной области;

— высокое качество обслуживания конечных пользователей;

— снижение стоимости разработки ИС и дальнейших инвестиций в развитие ПО;

— повышение производительности и эффективности использования;

— увеличение срока эксплуатации системы;

— объединение и рационализация бизнес-процессов;

— снижение риска, связанного с внедрением новых ИС;

— безопасное взаимодействие между пользователями в реальном времени;

— снижение времени выхода программного продукта на рынок;

— масштабируемость и гибкость.

Недостатки:

— сложность управления сервисами и организации их взаимодействия между собой;

— сложность тестирования сервисов, как самостоятельных, независимых от потребителя компонентов;

— несоответствие механизмов обеспечения безопасности в приложениях и сервисах;

— чем больше предметных областей охватывает сервис, тем сложнее вносить в него изменения.

На рисунке 1.3.2 представлен пример одной и той же системы, построенной на двух различных архитектурах. Классический вариант системы представляет собой единое монолитное приложение, внутри которого можно выделить ряд тесно связанных подсистем, выстроенных вокруг конкретных больших бизнес-процессов. Хотя архитектура такой системы и представляет собой трехслойную архитектуру, ее высокая связность значительно осложняет сопровождение и внедрение нового функционала в такую систему. Кроме того, реализация нового бизнес-процесса приводит к разработке новой подсистемы, большая часть исходного кода дублирует код уже имеющихся подсистем.

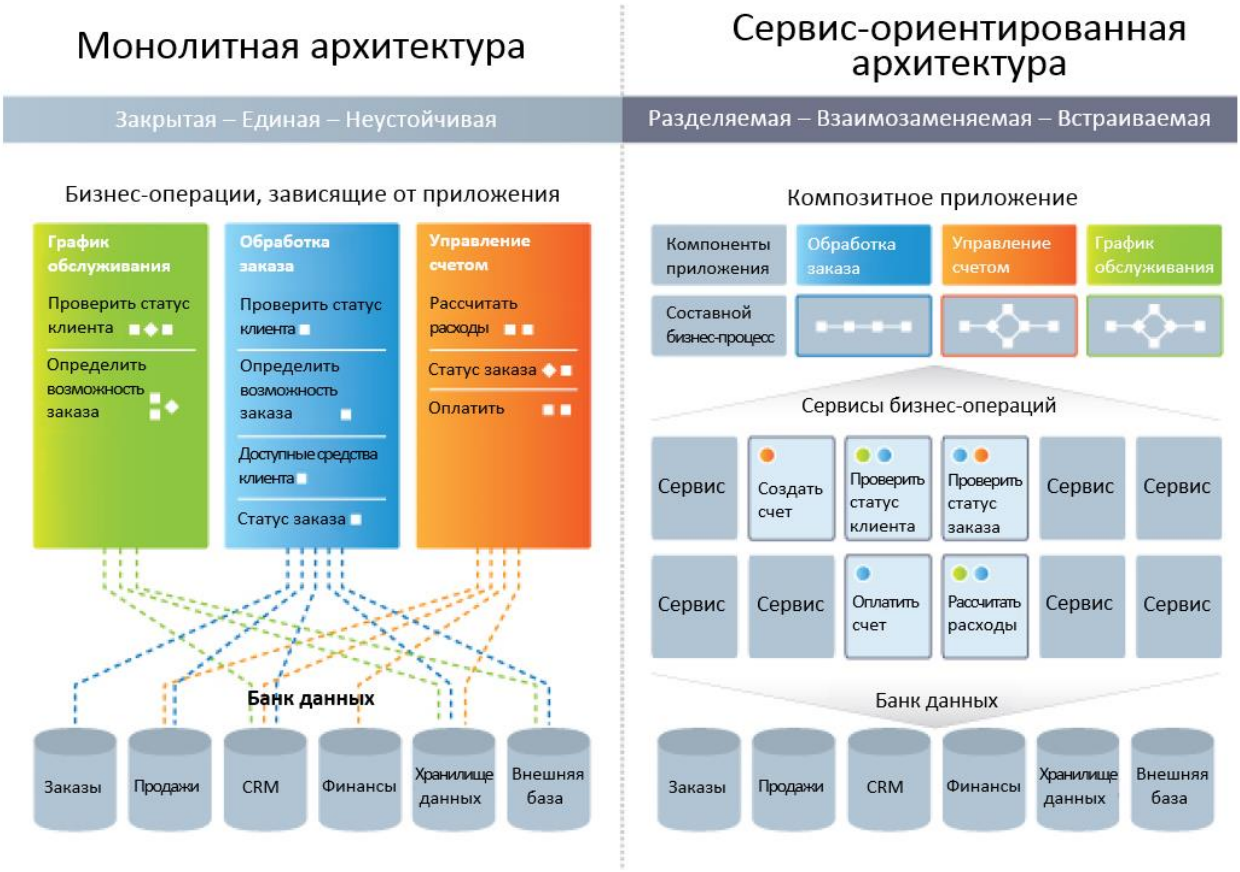

Рисунок 1.3.2 – Сравнение архитектур

Выделение базовых элементов из системы, а также разбиение бизнес-процессов на атомарные операции и их перенос в набор сервисов, не зависящих от реализации остальных компонентов системы, позволяет построить более гибкую систему на основе композитной архитектуры. При этом доступ к данным осуществляется через единый интерфейс, а доступ к сервисам может легко получить любая подсистема. К тому же такое разделение системы позволяет с меньшими трудозатратами выполнить разделение системы на несколько звеньев: сервер данных, сервер сервисов, клиентские приложения.

1.3.2 Композитная архитектура

Композитная архитектура – набор программных систем с множеством характеристик, которые удовлетворяют конкретным потребностям определенного сегмента рынка или выполняют определенную задачу, а также разрабатываются в установленном порядке и на основе общего набора базовых

средств. При этом операции интеграции и тестирования вытесняют операции проектирования и кодирования.

Построение информационных систем на основе композитной архитектуры обладает широким потенциалом:

— требования к программным продуктам, разработанным на основе композитной архитектуры, идентичны;

— проектирование архитектуры новой программной системы. Этот важнейший этап в разработке программного продукта может быть пропущен за счет использования уже готового проверенного решения;

— элементы. Повторное использование готовых проектных решений (например, элементы пользовательского интерфейса) позволяет избежать ошибок проектирования и сократить время разработки программной системы в целом;

— моделирование и анализ. Обладая набором уже готовых компонентов, моделирование и анализ новой системы значительно упрощается;

— тестирование. Повторное использование планов тестирования, наборов тестовых данных и отлаженных процессов тестирования;

— планирование проекта. Имеются уже готовые решения при составлении бюджета проекта, декомпозиции проектных задач, определении состава и размера групп разработчиков;

— процессы, методы, инструменты;

— специалисты. Возможность перебрасывать специалистов между проектами, область знания охватывает все программные продукты архитектуры;

— устранение дефектов. В каждой новой системе учитывается опыт устранения дефектов в предыдущих системах.

На рисунках 1.3.3 и 1.3.4 представлена информация об элементах и структуре композитной архитектуры, а также график экономической выгоды от использования композитных элементов, согласно исследованиям института [4].

Рисунок 1.3.3 – Композитная архитектура информационных систем

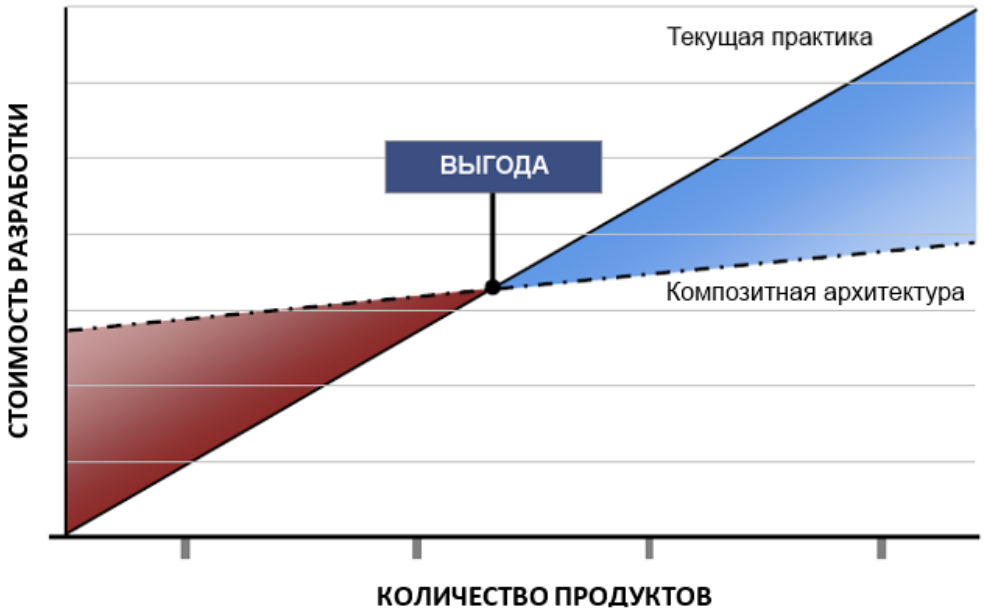

Рисунок 1.3.4 – Экономика композитной архитектуры

Разработка композитной архитектуры системы начинается с выделения и разделения всех ее элементов на две группы:

— неизменные элементы для всего семейства программных продуктов;

— изменяемые элементы, уникальные для каждой подсистемы.

Наиболее важным этапом в построении архитектуры композитного приложения является выявление изменяемых параметров системы. При этом можно выделить следующие подходы [5]:

Продуктивный (сначала разрабатывается набор базовых компонентов):

— сначала составляется техническое задание, затем на его основе ведется разработка;

— затраты на выпуск нового продукта на рынок минимальны;

— требуются инвестиции и прогнозирование рынка.

Данный метод применяется, когда заранее известно, что требуется разработка нескольких программных продуктов, а также известна большая часть требований к ним. Это позволяет выделить неизменные компоненты. В этом случае выход первого продукта задержится, но выход остальных будет намного быстрее.

Реактивный (сначала полностью разрабатываются несколько программных продуктов):

— на их основе формируется набор базовых компонентов для будущих продуктов;

— нет точного технического задания;

— минимальная стоимость ввода в эксплуатацию;

— более гибкая и расширяемая архитектура базовых компонентов, которая лучше всего отвечает потребностям будущих продуктов.

Такой подход применяется при поддержке важных и громоздких систем, стоимость поддержки которых ниже стоимости разработки новой системы.

Поэтапный (применяется к продуктивному или реактивному подходу):

— разрабатывается часть базовых компонентов;

— разрабатывается один или несколько продуктов;

— разрабатывается новая часть базовых компонентов;

— разрабатывается один или несколько продуктов;

— этот процесс повторяется для разработки каждого нового продукта.

Данный подход является наиболее оптимальным, так как позволяет быстрее выводить на рынок часть программных продуктов будущей системы и исходя из опыта их эксплуатации корректировать требования к следующим продуктам.

Не менее важным этапом в разработке композитных приложений является грамотное разделение задач и обязанностей по сферам профессиональной деятельности. Согласно [6] выделяют три области в жизненном цикле программного продукта: разработка приложения, техническое управление и организационный менеджмент. В таблице 1.3 представлено наиболее эффективное распределение задач по этим, решаемых при разработке программных продуктов на основе композитной архитектуры.

Таблица 1.3 – Направления деятельности

Разработка ПО	Техническое управление	Организационный менеджмент
Анализ архитектуры	Управление конфигурациями	Построение экономической модели
Разработка архитектуры	Сбор данных и измерения	Взаимодействие с пользователями
Разработка компонентов	Анализ себестоимости	Разработка стратегии продаж
Внедрение готовых компонентов	Определение рабочих процессов	Финансирование
Извлечение базовых компонентов	Анализ требований	Выпуск регламентов
Оценка требований к ПО	Техническое планирование	Анализ рынка
Интеграция ПО	Управление техническими рисками	Эксплуатация
Тестирование	Сопровождение	Организационное

		планирование
Определение предметной области		Управление организационными рисками
		Формирование структуры организации
		Планирование технического развития
		Обучение

Разработка каркаса композитной архитектуры основана на использовании ряда важных паттернов проектирования SOA. В соответствии с [7] и материалами с [8] наибольший интерес представляют, а также нашли практическое применение, следующие паттерны:

Транзакции атомарных сервисов. Если при выполнении сложного бизнес-процесса, обращающегося к нескольким сервисам, работа одного из сервисов завершается ошибкой, то помимо аварийного завершения всего бизнес-процесса могут быть повреждена или недействительна часть данных, которую успели обработать успешно завершенные сервисы. Для решения этой проблемы вводится объект-транзакция, в которую оборачиваются сервисы бизнес-процесса, и которая имеет возможность отмены и возврата всех изменений, сделанных сервисами внутри нее. Таким образом, любые изменения с данными выполняются только после того, как будут успешно завершены все сервисы бизнес-процесса. На рисунке 1.3.5 проиллюстрирована идея реализации паттерна.

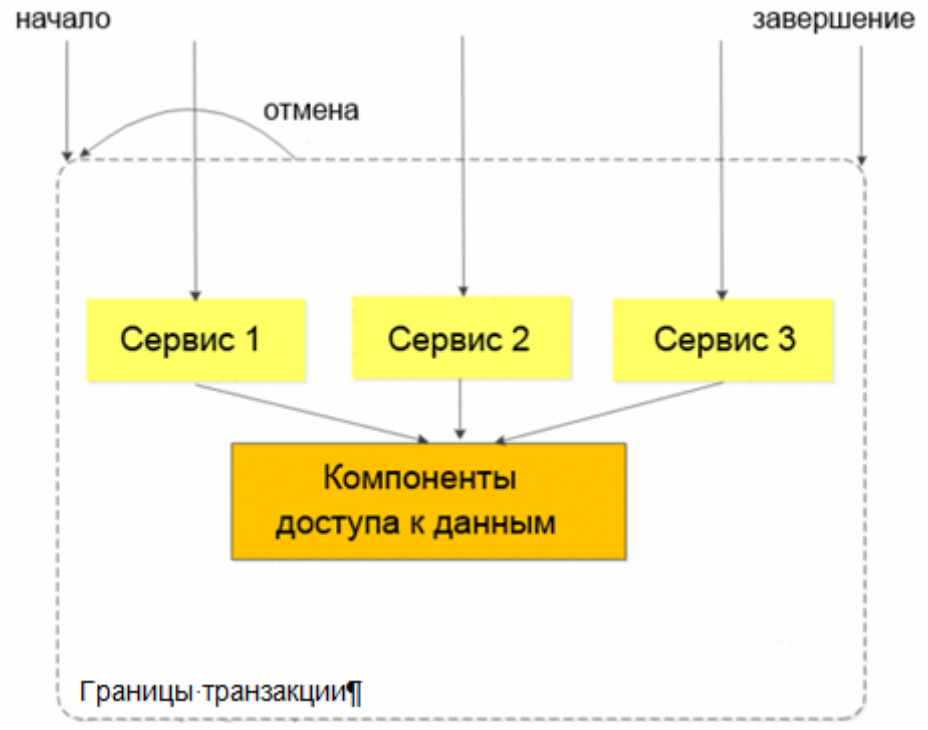

Рисунок 1.3.5 – Транзакция атомарных сервисов

В контексте композитной архитектуры объект-транзакция выносится в отдельный базовый компонент, на использовании которого в дальнейшем строятся все бизнес-сервисы архитектуры. Однако, из-за необходимости каждого сервиса сохранять свое состояние, возможно увеличение расхода памяти, поэтому важно выделять слой приложения (слой бизнес-логики) в отдельное звено (размещать на отдельном сервере).

Сервисная шина. Является одним из самых важных элементов сервис-ориентированной архитектуры. Играет роль посредника обмена сообщениями между сервисами и их потребителями. При этом может выполнять функции трансформации сообщений, распределения маршрутов между получателями, балансировки нагрузки на сервер и т.д. На рисунке 1.3.6 изображена схема использования сервисной шины.

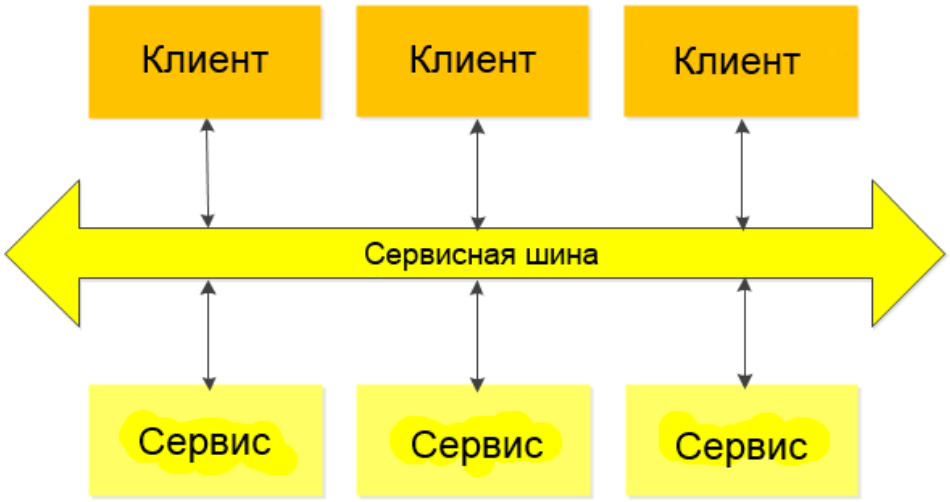

Рисунок 1.3.6 – Сервисная шина

Фасад сервисов. Служит для разрыва тесной связи между сервисами и их контрактами взаимодействия, тем самым сокращая количество изменений, которые нужно внести в сервис при изменении его контракта. Кроме того, каждый сервис может иметь несколько фасадов, тем самым обеспечивая реализацию нескольких различных контрактов. Однако такое усложнение архитектуры оправдано только при наличии в системе множества различных контрактов и, соответственно, сервисов. Схематичное представление паттерна показано на рисунке 1.3.7.

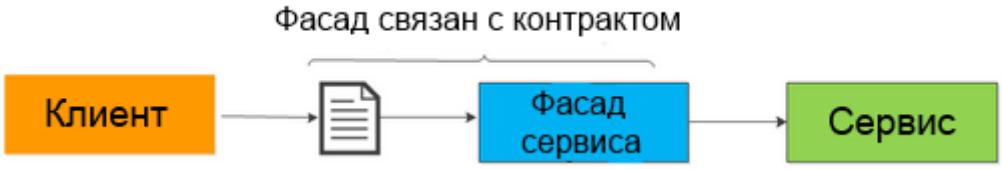

Рисунок 1.3.7 – Фасад сервиса

Посредник аутентификации. Не всегда имеется возможность использовать прямую аутентификацию, кроме того в рамках одного бизнес-процесса может понадобиться доступ к различным сервисам. Для обеспечения полноценной аутентификации вводится объект-посредник, который централизованно отвечает за аутентификацию клиентов сервисов, а также

раздачу специальных токенов (билетов) для упрощения процесса аутентификации клиентов между различными сервисами. Несмотря на то, что концентрация логики аутентификации в одной точке системы является потенциально уязвимым местом приложения, такой подход все же является более предпочтительным в контексте композитной архитектуры. На рисунке 1.3.8 показан принцип работы паттерна.

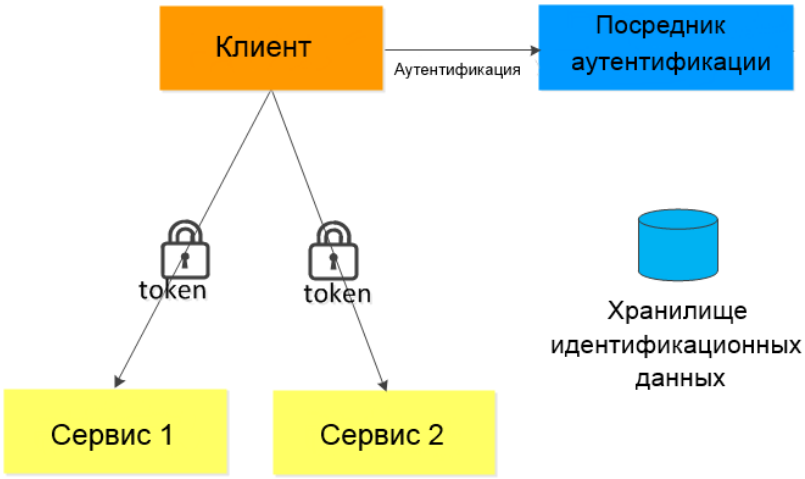

Рисунок 1.3.8 – Принцип работы посредника аутентификации

Экран сообщений. Является дополнительным средством обеспечения безопасности для случаев, когда злоумышленник пытается подменить или повредить сообщение, предназначенное сервису. Для этого все сообщения перенаправляются в специальный объект, который воспринимает все входящие сообщения вредоносными, пока не будет доказано обратное. Несмотря на незначительный рост времени отклика, увеличение устойчивости системы является большим преимуществом. На рисунке 1.3.9 показан принцип работы экрана сообщений.

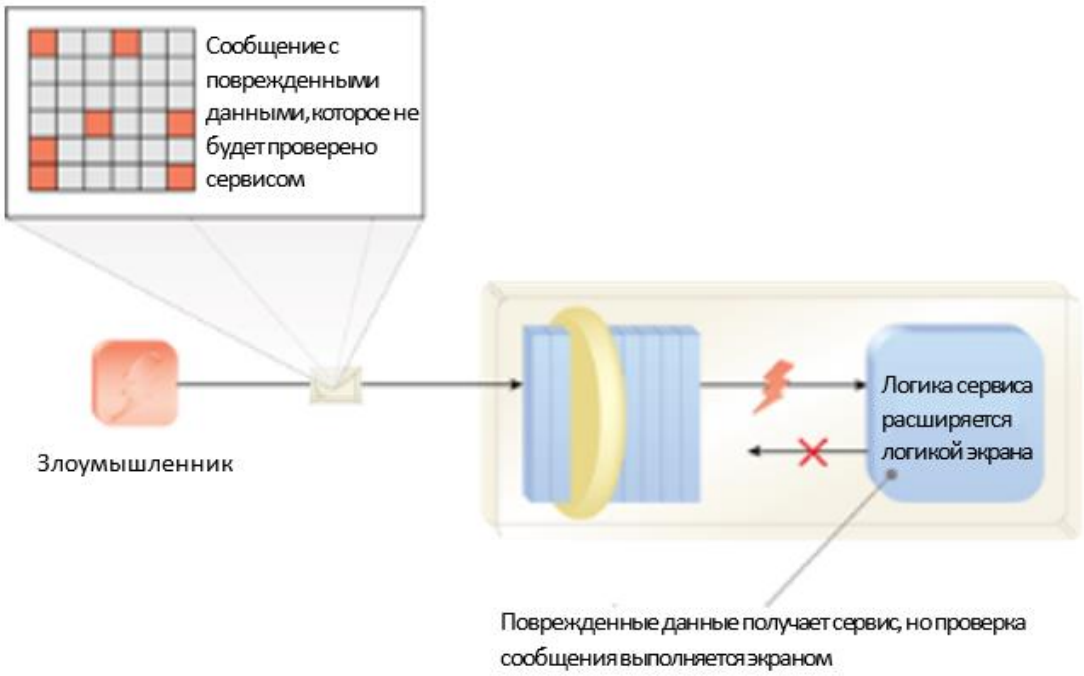

Рисунок 1.3.9 – Экран сообщений

Композитная сервис-ориентированная программная система представляет собой совокупность сервисов, сервисных компонентов и компонентов доступа к данным, которые проектируются и разворачиваются в одном приложении. Взаимодействие между сервисами и различными компонентами осуществляется путем обмена сообщениями. На рисунке 1.3.10 представлен пример реализации отдельного бизнес-процесса в композитном приложении [9].

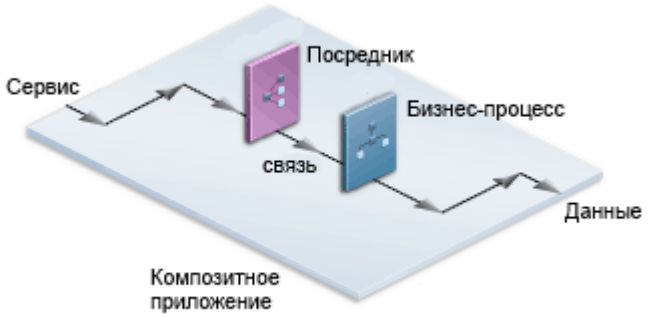

Рисунок 1.3.10 – Структура реализации бизнес-процесса

Сервисные компоненты. Являются строительными блоками при построении композитных сервис-ориентированных приложений. Каждый компонент размещается в определенном сервисном контейнере. Сообщения передаются в сервисные контейнеры, а затем внутри них перенаправляются в соответствующие сервисные компоненты. Кроме того, сервисные контейнеры могут взаимодействовать и между собой. Например, предназначенное для какого-либо бизнес-процесса сообщение сначала посылается в контейнер этого процесса. Затем сервисный контейнер обрабатывает полученную информацию и перенаправляет сообщения на соответствующие сервисные компоненты внутри себя.

Сервисные компоненты решают следующие задачи:

— описывают и реализуют бизнес-процессы предметной области;

— задают бизнес-правила;

— описывают рабочие процессы на основе бизнес-процессов;

— передают сообщения между компонентами.

Сервисы. Обеспечивают взаимодействие между композитным приложением и его потребителями. Связь с сервисом осуществляется по заданным протоколам (например, SOAP/HTTP).

Компоненты доступа к данным. Служат для извлечение и модификации данных на основе сообщений, переданных от сервисных компонентов.

Связи. Описывают способы обмена информацией между сервисами и сервисными компонентами, между различными сервисными компонентами или между сервисными компонентами и компонентами доступа к данным.

В программных системах уровня предприятия, как правило, выделяют три архитектурных слоя: слой представления, слой приложения (или бизнес-логики) и слой данных. Соответственно, для каждого слоя должен быть разработан свой сервисный контейнер. Большая часть требований к контейнерам уже известна на этапе проектирования основных слоев приложения. Таким образом из наборов сервисных компонентов и контейнеров можно собрать различные варианты программной системы, не затрачивая при

этом усилий на разработку и интеграцию. Главное отличие и преимущество композитной архитектуры перед SOA в том, что композитная архитектура обеспечивает гибкость на всех слоях (уровнях) приложения, в то время, как SOA предоставляет гибкость только на одном слое – слое приложения (бизнес-логики) [10]. На рисунке 1.3.11 показано абстрактное представление композитной архитектуры приложения уровня предприятия.

На самом верху архитектуры находятся обработчики информации (англ., information workers), которые предоставляют доступ к бизнес-данным посредством порталов (специальные презентационные средства уровня предприятия). Также обработчики служат для формирования специальных документов, как результата бизнес-операций, являющихся частью больших бизнес-процессов. Бизнес-процессы координируют действия пользователей и системы. Для управления системными операциями служат бизнес-правила, которые запускают низкоуровневые службы посредством сервисных интерфейсов. Операции пользователей встраиваются в бизнес-процессы с помощью событийной модели. После применения всех бизнес-правил и событий к сформированным документам бизнес-процесс завершается.

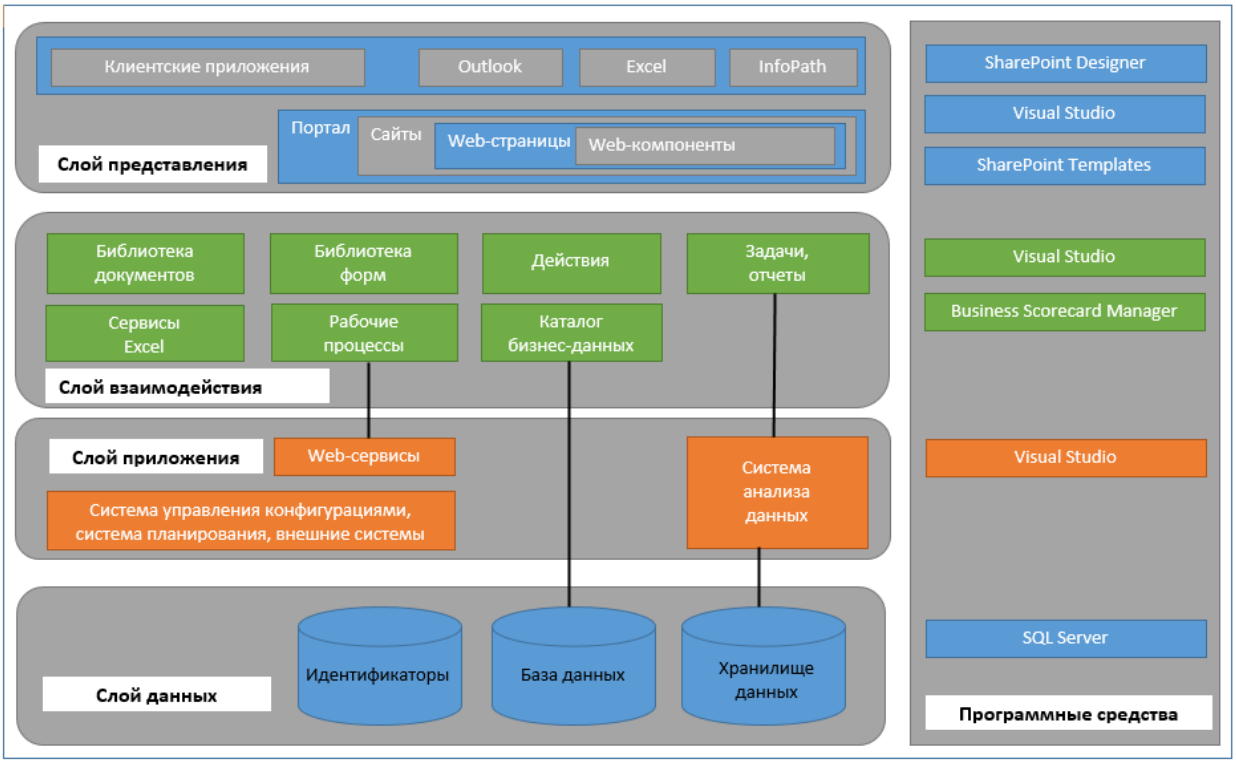

Рисунок 1.3.11 – Пример архитектуры композитного приложения

Так как композитная архитектура по своей сути является дальнейшим развитием SOA и, соответственно, обладает всеми преимуществами сервис-ориентированного подхода в разработке программного обеспечения. Однако помимо этого в композитных приложениях имеется и ряд дополнительных плюсов:

— с точки зрения разработчика приложения, это проектирование, реализация и разворачивание приложений уровня предприятия на протяжении всего времени жизни приложений;

— с точки зрения пользователей, это готовая комплексная бизнес-единица, решающая весь спектр необходимых задач предметной области.

Главным недостатком служит то, что крайне затруднительно разработать качественное и экономически эффективное композитное приложение «с нуля». Композитные приложения, как правило, являются дальнейшим развитием уже разработанной группы отдельных приложений, из которых выделяются базовые наборы компонентов и контейнеров для их дальнейшей интеграции друг с другом.

В настоящее время идея построения сервис-ориентированных приложений на основе композитной архитектуры находит широкое распространение в различных областях разработки программного обеспечения и является предметом изучения во многих научно-исследовательских центрах. Одной из самых крупных подобных организаций является Институт инженерии программного обеспечения (англ., Software Engineering Institute, SEI) [4]. Институт ведет научные исследования по поиску решений проблем инженерии программного обеспечения, систематизирует технологические и методологические решения, проводит тестирование экспериментального ПО, а также осуществляет обучение и лицензирование организаций. На данный момент институт является признанным международным лидером в изучении вопроса построения композитных сервис-ориентированных приложений.

Методология, разработанная институтом, служит основой при переходе от классических монолитных систем к композитным.

1.4 Вывод

Из вышеперечисленных подходов мы выбрали сервис-ориентированный, так как он подразумевает, что наше приложение будет состоять из набора независимых сервисов, каждый из которых реализует отдельную бизнес-функцию, которая является логически обособленной, повторяющейся задачей, являющейся составной частью бизнес-процесса предприятия. Более того, сервисы могут быть реализованы независимо от языков программирования и других технических особенностей реализации, что дает возможность использовать различные технологии и фреймворки. Также сервисы могут быть написаны в независимости от других служб системы, необходимо только знание интерфейса используемых сервисов, то есть службы будут слабосвязны (loose coupling).

Мы прогнозируем изменения в КИС в ближайшее время. Например, учиться будут не только студенты, но и работники предприятий. В перспективе допустим выход системы как продукта (система для повышения квалификации). Поэтому выбор пал на сервис-ориентированную архитектуру.

Иными словами, мы выделили следующие достоинства данного подхода:

— отсутствие дублирования кода;

— независимость от языка программирования;

— низкая связанность.

2 Практическая часть

2.1 Инструментальные средства разработки

Разработав техническое задание была построена диаграмма Ганта, которая позволила оценить временные затраты на разработку. Она представлена на рисунке 2.1.1

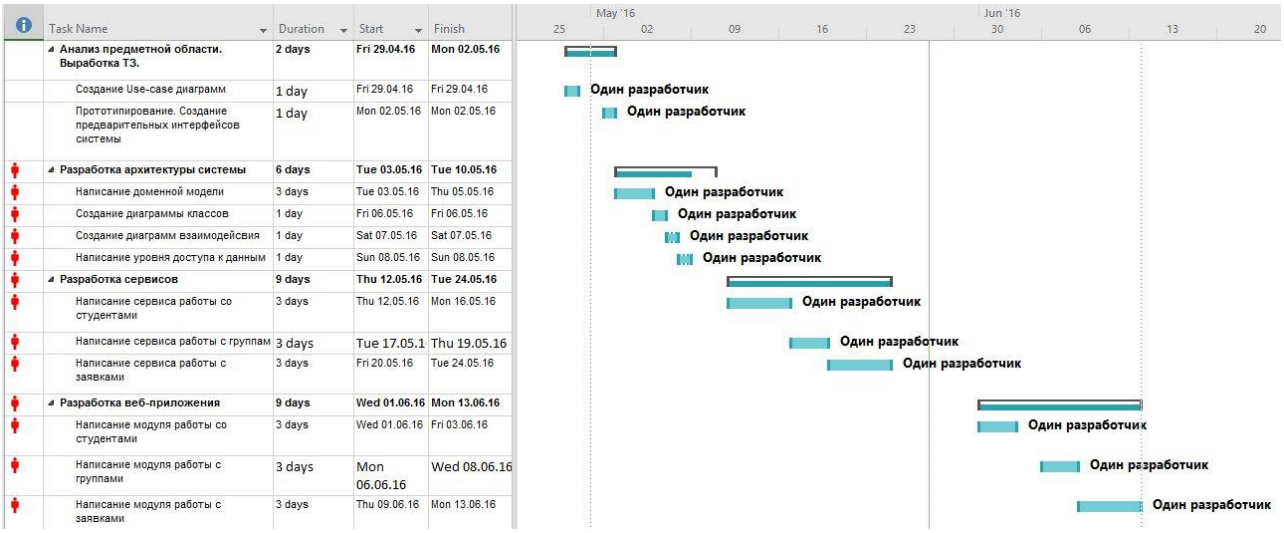

Рисунок 2.1.1 – Диаграмма Ганта

График трудовых затрат для разработчика по дням представлен на рисунке 2.1.2.

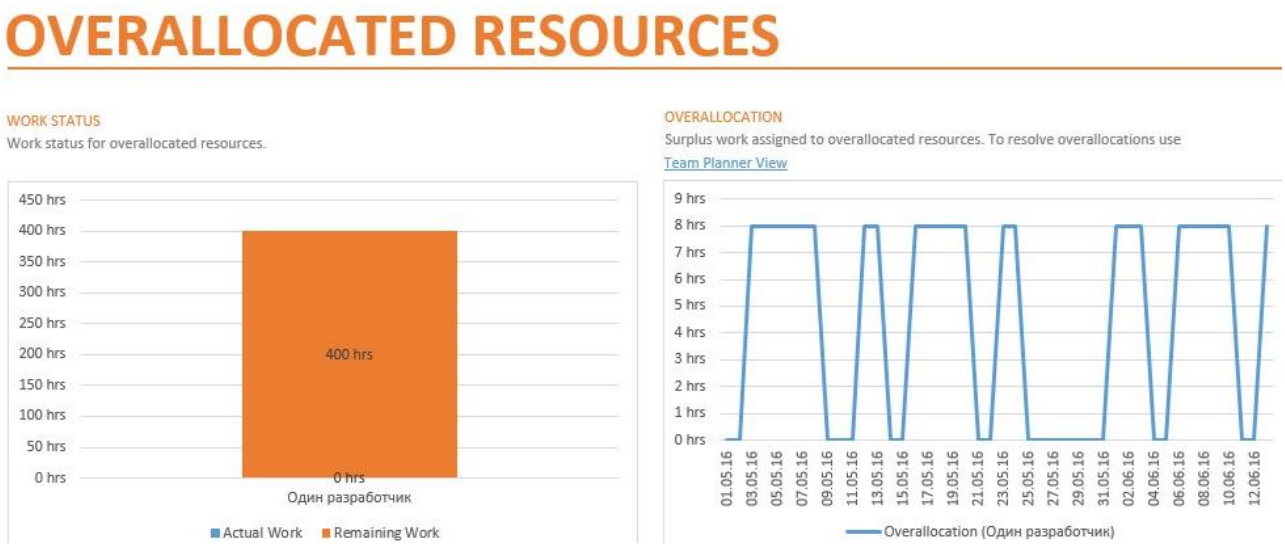

Рисунок 2.1.2 – График трудовых затрат для одного разработчика

Проанализировав временные затраты был сделан вывод о том, что в рамках выполнения выпускной квалификационной работы бакалавра справиться в срок одному программисту не является возможным. Поэтому было привлечено два разработчика:

— Костюк Александр Владиславович;

— Васильев Эдуард Владимирович.

На рисунке 2.1.3 представлена диаграмма Ганта с учетом двух разработчиков.

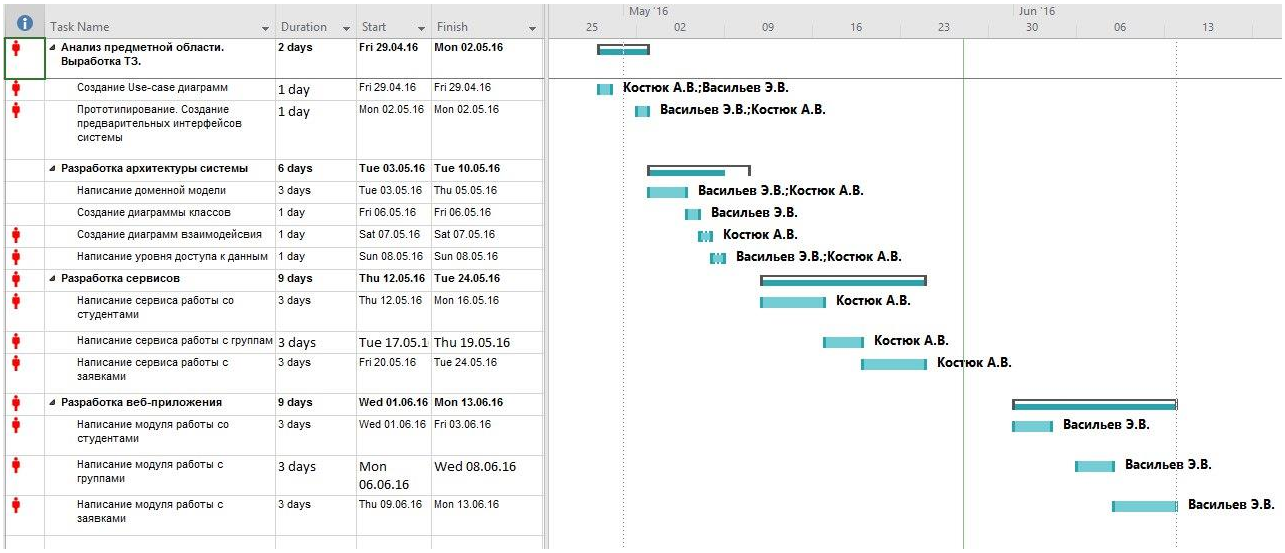

Рисунок 2.3 – Диаграмма Ганта с учетом двух разработчиков

График трудовых затрат для двух разработчиков по дням представлен на рисунке 2.1.4

Рисунок 2.1.4 - График трудовых затрат для двух разработчиков

Для командной разработки необходим такой инструмент как система управления версиями. Она позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое. Для этой задачи был выбран TFS (Team Foundation Server), т.к. разработка ведется в IDE (Integrated Development Environment) Visual Studio 2015, где TFS доступен по умолчанию, что упрощает интеграцию системы управления версиями и среды разработки. На рисунке 2.1.5 представлено окно обозревателя исходного кода в Visual Studio 2015.

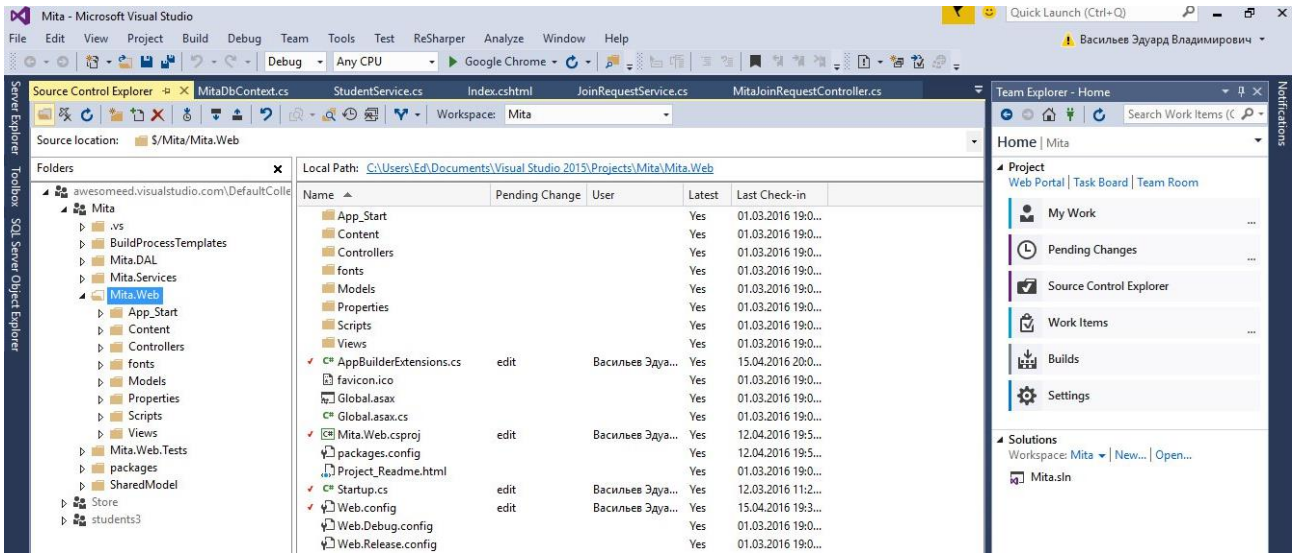

Рисунок 2.1.5 – Source Control Explorer

Так как проект реализуется несколькими разработчиками, то возникает необходимость использования методологии ведения разработки. Команда разработчиков новая, интенсивность работы каждого неопределенна, поэтому не является возможным рассчитать, сколько именно времени понадобится на реализацию той или иной задачи. Для этого случая отлично подходит методология Kanban, основной задачей которой является уменьшение количества «выполняющейся в данный момент работы» (work in progress). Ведь не исключена возможность того, что задачи могут быть выполнены как раньше, так и позже поставленного срока, следовательно, рабочие ресурсы будут распределяться более рационально путем смещения разработчиков с одной выполненной задачи на другую, которая отстает по срокам. На рисунке 2.1.6 представлен основной инструмент данной методологии – Kanban доска.

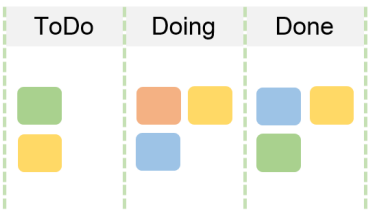

Рисунок 2.1.6 – Kanban доска

2.2 Архитектура и процесс разработки

2.2.1 Технология ASP.NET

Для реализации поставленной задачи была выбрана технология создания веб-приложений и веб-сервисов от компании Майкрософт - ASP.NET, включающая в себя MVC паттерн. Это было обусловлено тем, что в ASP.NET MVC доступна функция автогенерации представлений и контроллеров, что, в какой-то степени, ускоряет разработку, учитывая временные ограничения ВКР.

Концепция паттерна (шаблона) MVC (model - view - controller) предполагает разделение приложения на три компонента:

— **контроллер (controller)** представляет класс, обеспечивающий связь между пользователем и системой, представлением и хранилищем данных. Он получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления;

— **представление (view)** - это собственно визуальная часть или польз овательский интерфейс приложения. Как правило, html-страница, которую пользователь видит, зайдя на сайт;

— **модель (model)** представляет класс, описывающий логику используемых данных.

Общая схема взаимодействия этих компонентов представлена на рисунке 2.2.1:

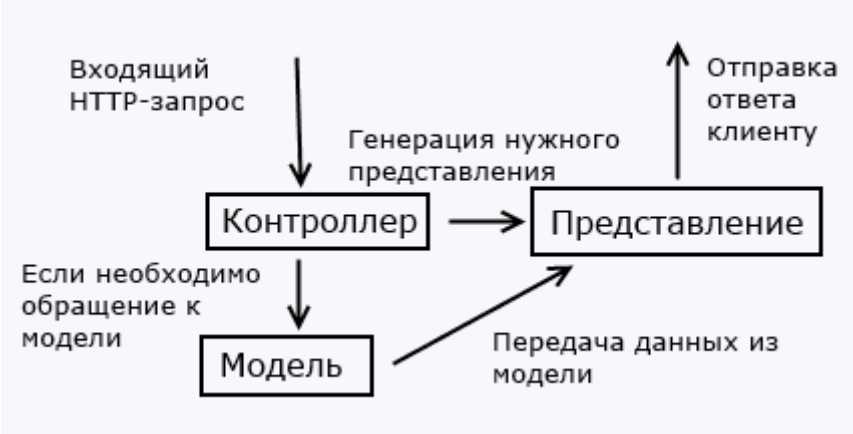

Рисунок 2.2.1 - Паттерн MVC в ASP.NET

В этой схеме модель является независимым компонентом - любые изменения контроллера или представления не затрагивают модель. Контроллер и представление являются относительно независимыми компонентами, и нередко их можно изменять независимо друг от друга.

Благодаря этому реализуется концепция разделение ответственности, в связи с чем легче построить работу над отдельными компонентами. Кроме того, вследствие этого приложение обладает лучшей тестируемостью. И если нам, допустим, важная визуальная часть или front-end, то мы можем тестировать представление независимо от контроллера. Либо мы можем сосредоточиться на back-end и тестировать контроллер. В силу своей гибкости и простоты он стал очень популярным в последнее время, особенно в сфере веб-разработки.

Также в ASP.NET MVC создана мощная система маршрутизации, которая упрощает представление URL ссылок, что позволяет, во-первых, придать поисковым системам значительный вес ключевым словам, находящимся в URL. Во-вторых, многим пользователям Интернета теперь хватит навыков и знаний, чтобы понять URL, и оценить возможности навигации, набрав его в адресной строке своего браузера. В-третьих, когда кто-то понимает структуру URL, он, скорее всего, будет ссылаться именно на него, поделится этой ссылкой с другом или даже продиктует ее вслух по телефону. В-четвертых, такая ссылка не предоставляет технические подробности, папки, имена файлов и структуру приложения на весь общественный Интернет, так что вы можете изменить внутреннюю реализацию, не нарушая ссылки.

По умолчанию ASP.NET MVC используется движок представления Razor. для ориентированных на код шаблонов. Процесс программирования становится более быстрым, синтаксис является компактным и лаконичным, в то же время улучшая читабельность вашей разметки и кода.

Но самой главной возможностью у ASP.NET, является замена буквально любой части кода. Если определенный механизм не удовлетворяет вашим потребностям, вы можете легко заменить его на свою реализацию. В проекте

используются технологии, предоставляемые ASP.NET по умолчанию. Однако, это не это не мешает включить в проект новые технологии или заменить старые. К тому же, это не составит больших трудозатрат, т.к. вся логика системы реализуются в сервисах и не завязаны на реализацию приложения.

2.2.2 ORM Entity Framework

В своем проекте мы решили использовать ORM (Object Relational Mapping). Это было обосновано тем, что у нас не было времени писать SQL-запросы вручную, а также, ORM позволяет создавать более лучший код. В конечном итоге, проект будет проще сопровождать.

Перед нами встал выбор между Entity Framework и NHibernate. Мы провели сравнительный анализ этих двух ORM:

Nhibernate более сложная в изучении по сравнению с EF. Однако мы не нашли в нашем проекте ни одной задачи, с которой бы не справился Entity Framework. В добавок к этому, мы имели с ним опыт работы.

В нашем проекте нам необходимо использовать асинхронные методы. EF поддерживает асинхронные операции, в то время, как NHibernate не оставляет такой возможности.

Для нас встроенная поддержка миграции данных – это необходимая возможность для ORM, т.к. модель данных меняется. Соответственно, необходимо в след за ней менять базу данных. Это стало одним из решающих факторов, почему мы выбрали Entity Framework.

Если рассматривать данную ORM глубже, то Entity Framework представляет собой набор технологий ADO.NET, обеспечивающих разработку приложений, связанных с обработкой данных. Платформа позволяет разработчикам создавать приложения для доступа к данным, работающие с концептуальной моделью приложения, а не напрямую с реляционной схемой хранения. Цель состоит в уменьшении объема кода и снижении затрат на сопровождение приложений, ориентированных на обработку данных [11].

Технология схожа с LINQ to SQL, но имеет некоторые отличия. Entity Framework, в отличие от LINQ to SQL, поддерживает работу с СУБД, отличными от MS SQL Server, в частности, с Oracle Database и DB2. Технологию можно сопоставить с паттерном Domain Model (модель области определения), описанный М. Фаулером [12]. Кроме этого, LINQ to Entities позволяет использовать модель сущностей, которая значительно отличается от соответствующей модели данных, в то время как LINQ to SQL создает жесткую привязку генерируемых классов к таблицам базы данных (одна таблица – один класс). Фактически, структура модели LINQ to Entities не привязана к источнику данных – необходимо лишь проставить соответствия свойства генерируемых классов с реальными столбцами таблиц или представлений БД. Вместе с тем, LINQ to SQL позволяет использовать собственные классы сущностей, наследоваться от интерфейсов или базового класса сущностей. В Entity Framework такой возможности нет.

Платформа Entity Framework позволяет работать с данными в форме специфических для домена объектов и свойств, таких как клиенты и их адреса, без необходимости обращаться к базовым таблицам и столбцам базы данных, где хранятся эти данные. Entity Framework дает разработчикам возможность работать с данными на более высоком уровне абстракции; создавать и сопровождать приложения, ориентированные на данные, используя меньше кода, чем в традиционных приложениях. Поскольку Entity Framework является компонентом .NET Framework, приложения Entity Framework могут работать на любом компьютере, где установлена платформа .NET Framework, начиная с версии 3.5 с пакетом обновления 1 (SP1).

Основные возможности платформы [11]:

— поддерживает работы с различными серверами баз данных (MS SQL Server, Oracle, DB2);

— позволяет строить модель по схемам физических баз данных, а также хранимым процедурам;

— интегрируется со средой разработки Visual Studio, для визуальной и автоматической генерации модели по имеющимся базам данных;

— позволяет сгенерировать модель, а также базу данных на ее основе по семантическому описанию (подход Code First);

— успешно сочетается с другими технологиями разработки сервис-ориентированных приложений (ASP.NET, WPF, WCF, WCF Data Services).

На рисунке 2.2.2 показана роль Entity Framework в приложении.

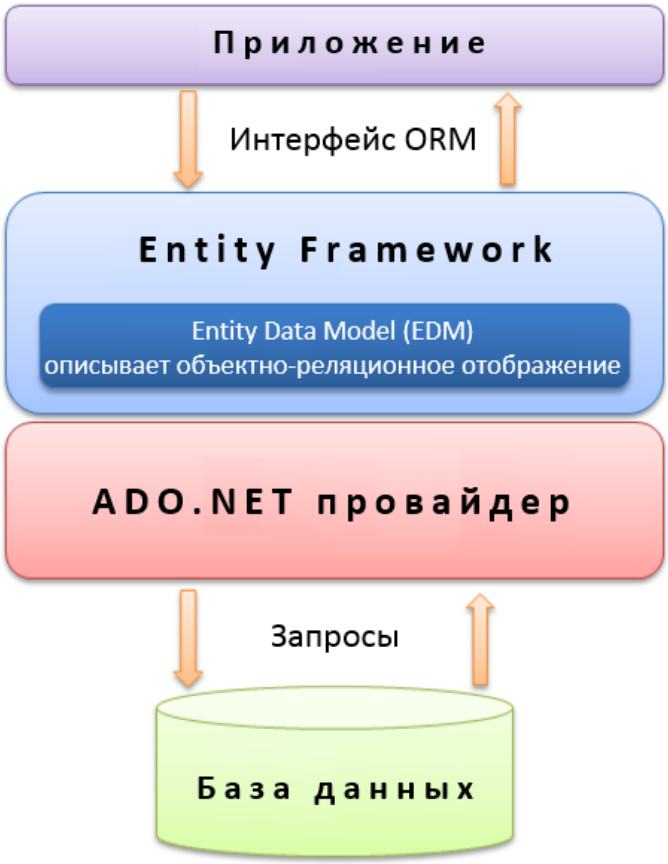

Рисунок 2.2.2 - Место Entity Framework в приложении

Можно сделать вывод, что использовать Entity Framework разумно в больших проектах со сложной структурой данных, а также в случаях, когда необходима поддержка сторонних СУБД (не MS SQL Server).

Как показывает опыт, времени на проектирование и написание кода, как правило, всегда мало – и архитекторы, и разработчики ищут пути наименьшего сопротивления. В большинстве ситуаций технология ADO.Net себя не

оправдывает, и её применение в некоторых проектах обусловлено совместимостью с ранее написанным кодом. Однако в силу достаточного большого возраста технологии, у разработчиков накоплено значительное количество наработок – фреймворков, оберток, вспомогательных классов, применение которых несколько сглаживает недостатки ADO.Net. В новых же проектах всегда используются технологии LINQ to SQL или Entity Framework, в которых возможность автоматического создания классов из сущностей БД значительно уменьшает время разработки, сокращает размер написанного кода, а также делает его более понятным. Хорошо зарекомендовал себя комбинированный способ – на ASP.Net сайте системы использовать Entity Framework, т. к. часто необходимо создавать сложные сущности на основе уже имеющихся, и очень удобно редактировать модель прямо в ORM-дизайнере. Напротив, для служб, которые выполняют в фоне анализ данных и работают напрямую с БД, проще обойтись технологией LINQ to SQL, сущности которой жестко привязаны к сущностям базы.

Главные преимущества, которые дает использование Entity Framework в разработке композитных сервис-ориентированных приложений [13]:

— снижает время разработки, позволяя уделять больше времени логике приложения, а не способам взаимодействия с данными;

— взаимодействие с данными осуществляется по концептуальной модели, а не конкретной структуре хранения самих данных, что в свою очередь позволяет разрабатывать базовые компоненты, независимые от данных, с которыми они работают;

— позволяет избежать жестко прописанных зависимостей обработки данных в коде приложения, а также использования хранимых процедур;

— изменения в схеме данных автоматически применяются к объектно-реляционной модели этих данных, генерируемой платформой Entity Framework;

— гибкая поддержка языка интегрированных запросов (Language-Integrated Query, LINQ), что позволяет автоматизировать операции группировки, сортировки и фильтрации данных.

2.2.3 IoC-контейнер Autofac

Инверсия управления (англ. Inversion of Control, IoC) – один из самых важных принципов объектно-ориентированного программирования, используемый для уменьшения связанности в компьютерных программах. Суть принципа в том, что модули верхнего и нижнего уровня должны зависеть от абстракций, а не друг от друга, а сами абстракции не должны зависеть от деталей реализации.

В данном случае IoC-контейнер – это компоновщик, который собирает между собой объекты (экземпляры классов) программы во время ее исполнения по мере обращения к объектам, а не на этапе компиляции. Одной из реализаций компоновщика является контейнер Autofac. В текущей реализации контейнера зависимостей поддерживаются инъекции конструкторов, свойств и методов.

Использование IoC-контейнера дает следующие преимущества в разработке любых крупных приложениях:

— снижение стоимости поддержки исходного кода;

— снижение сложности автоматизированного Unit-тестирования, а также возможность применять принцип разработки через тестирование (Test Driven Development, TDD);

— гибкость и расширяемость;

— связывание объектов на этапе выполнения программы;

— параллельная разработка, за счет отделения функциональных элементов друг от друга;

— устранение сквозной функциональности (функционал, общий для разных частей/модулей программы, например, обработка исключений или формирование журнала событий);

— снижение связности объектов.

В контексте композитных приложений использование IoC-контейнера позволяет выделять базовые независимые друг от друга элементы, легко

заменяемые, как на этапе разработки, так и во время выполнения готового приложения. Эта возможность позволяет довольно гибко конфигурировать приложение без необходимости вносить изменения в исходный код.

2.3 Структура проекта

Для реализации сервис-ориентированной архитектуры был немного изменен классический патер MVC. На рисунке 2.3.1 представлена диаграмма последовательностей классического паттерна MVC в ASP.NET

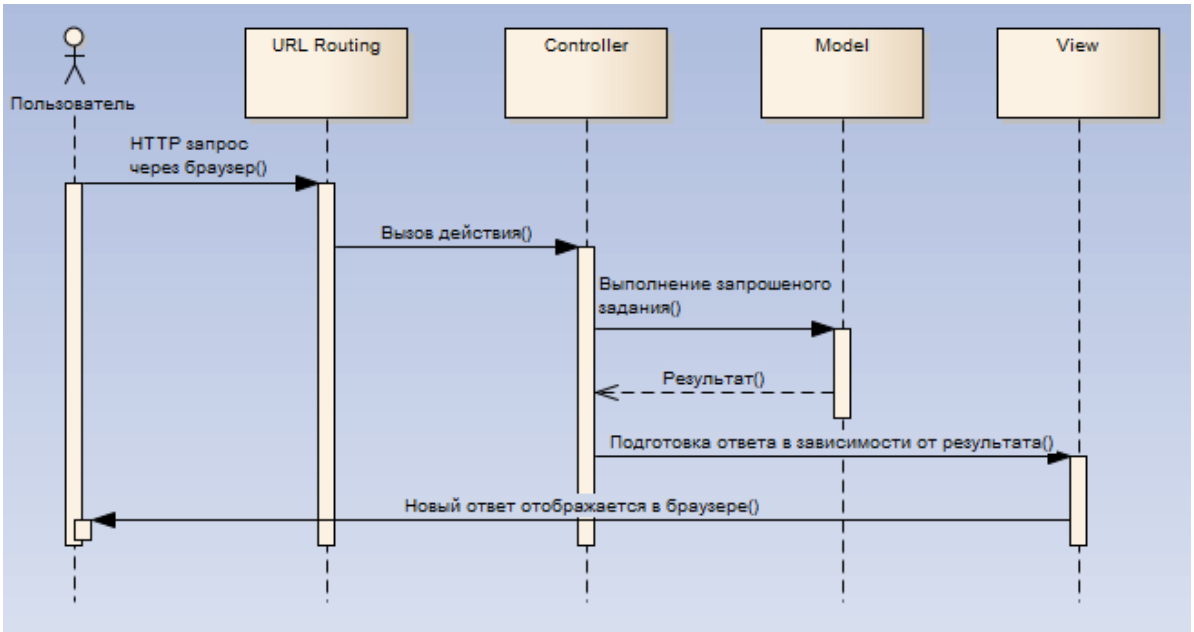

Рисунок 2.3.1 – Диаграмма последовательностей MVC

На рисунке 2.3.2 представлена диаграмма последовательностей с использованием сервисов.

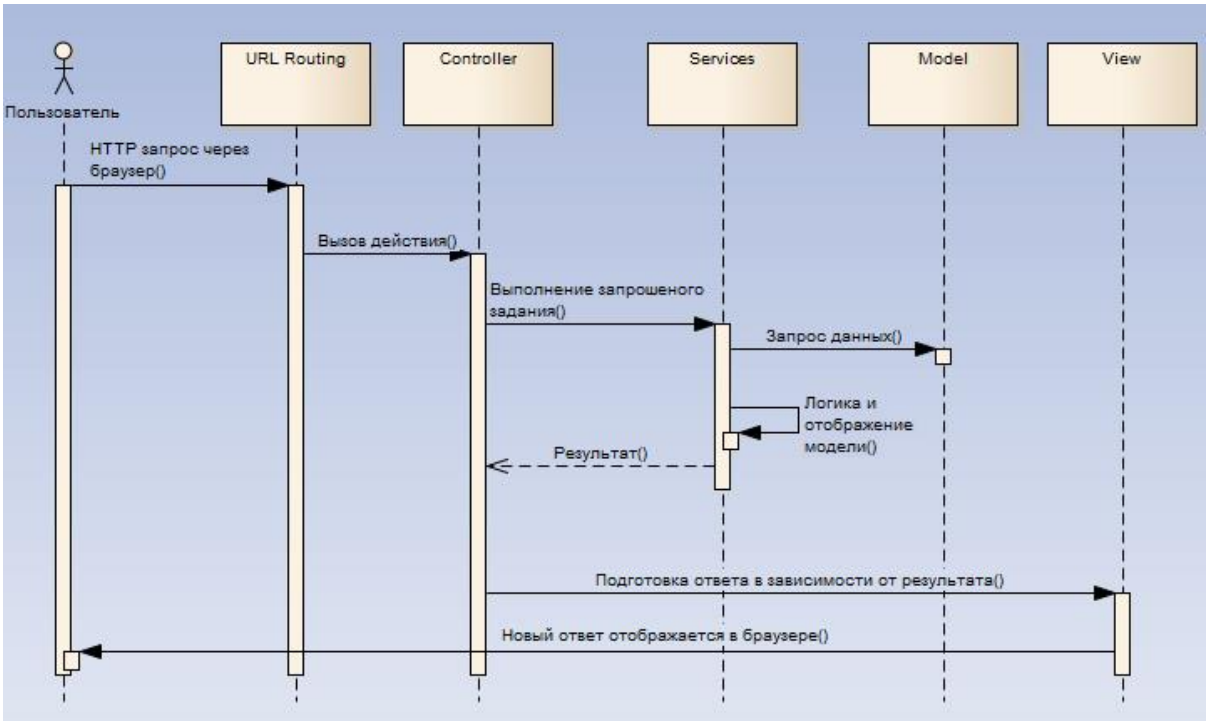

Рисунок 2.3.2 – Диаграмма последовательностей с использованием сервисов

Таким образом, решение информационной системы MitaIS состоит из пяти проектов. Особенность данной архитектуры заключается в том, что слои не зависят друг от друга. Это значит, что возможно изменить или заменить один из слоев, что некритически скажется на других слоях. Например, можно изменить систему представлений или редактировать бизнес-логику в сервисах. Структура решения представлена на рисунке 2.3.3.

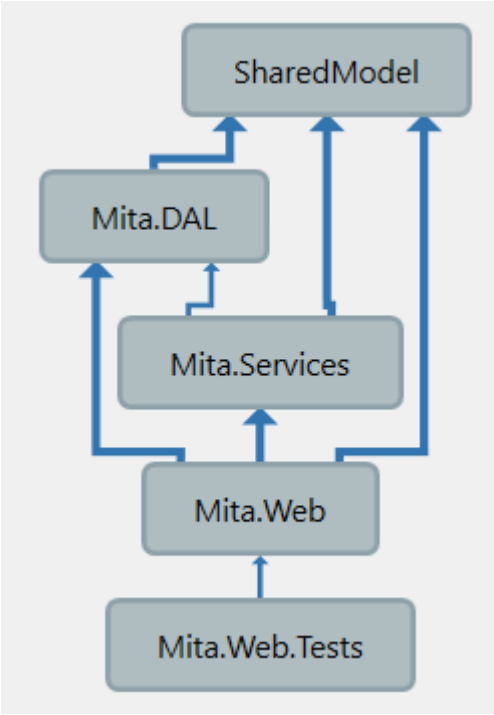

Рисунок 2.3.3 – Структура проектов информационной системы

Проект SharedModel содержит в себе доменную модель. Её диаграмму классов можно увидеть на рисунке 2.3.4.

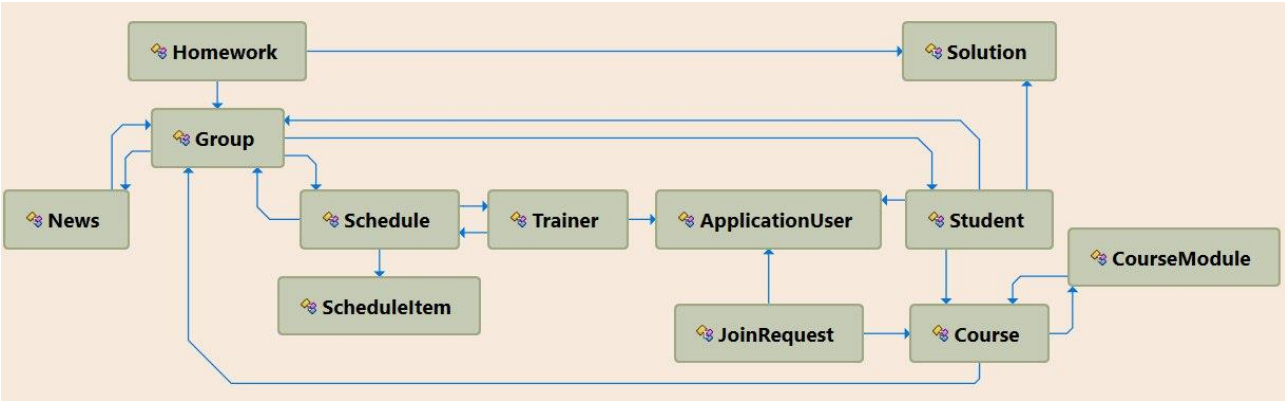

Рисунок 2.3.4 – Диаграмма классов

Mita.DAL – проект, отвечающий за уровень доступа к данным. Данный проект служит для связи между приложением и базой данных. Класс MitaDbContext описывает таблицы базы данных приложения. А класс MitaDbContextPorvider содержит в себе методы для работы с данными,

полученными из контекста. Структура проекта Mita.DAL представлена на рисунке 2.3.5.

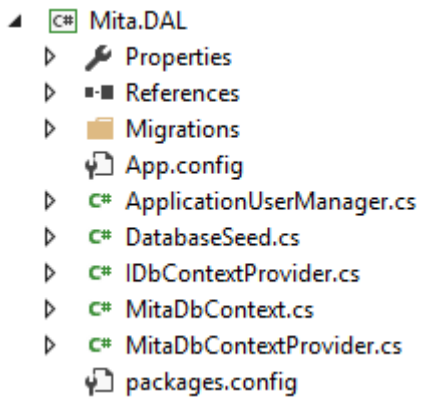

Рисунок 2.3.5 – Структура проекта Mita.DAL

Проект Mita.Services содержит в себе основную бизнес-логику приложения. А также, в проекте расположены модели сущностей, созданные специально для отображений. Наличие этих моделей обусловлено тем, что пользователю не обязательно видеть некоторые поля объектов, находящихся в базе данных. И напротив, некоторые поля из представлений не должны храниться в БД.

Mita.Web является основным проектом системы, к которому подключаются все сервисы. Данный проект представляет собой приложение ASP.NET MVC.

Проект Mita.Web.Test служит для написания Unit-тестов для приложения.

2.4 Реализация

Для разработки веб-приложения я использовал уже реализованные ранее сервисы. Подробнее об их реализации можно ознакомиться в выпускной квалификационной работе Александра Костюка из группы КИ12-13Б.

В первую очередь необходимо настроить IoC-контейнер. Для этого был создан класс AppBuilderExtensions. В нем регистрируются все сервисы, которые

будут использоваться. Конкретная реализация класса заменяется на интерфейс, который этот класс реализует. Это обеспечивает возможность писать код, используя ещё не до конца реализованные сервисы. А также, упрощает сопровождение проекта, так как при изменении логики какого-либо сервиса, не придется вносить изменения в код самого веб-приложения. На рисунке 2.4.1 показана регистрация сервисов в IoC-контейнере.

```
// Services
builder.RegisterType<StudentService>().As<IStudentService>();
builder.RegisterType<JoinRequestService>().As<IJoinRequestService>();
builder.RegisterType<CourseService>().As<ICourseService>();
```

Рисунок 2.4.1 – Регистрация сервисов

После того, как сервисы зарегистрированы, можно перейти к написанию контроллеров. В каждом контроллере сперва реализуется паттерн «инъекция зависимости» (англ. Dependency injection, DI). Это производится для создания объектов, реализующих сервисы. На рисунке 2.4.2 представлена инъекция зависимости для контроллера MitaJoinRequestController, в котором потребовались функции сервисов работы с заявками и работы с курсами.

```
private readonly IJoinRequestService _joinRequestService;
private readonly ICourseService _courseService;


0 references
public MitaJoinRequestController(IJoinRequestService joinRequestService, ICourseService courseService)
{
    _joinRequestService = joinRequestService;
    _courseService = courseService;
}
```

Рисунок 2.4.2 - Инъекция зависимости

Затем в проект была добавлена библиотека ASP.NET Identity. Она обеспечивает широкий инструментарий для работы с пользователями. Сперва была реализована авторизация пользователей через социальную сеть. Для этого было создано приложение в самой социальной сети. Там были получены ID

приложения и секретный ключ. Они копируются в класс Startup. На рисунке 2.4.3 изображен листинг кода класса Startup, в котором показано, как конфигурируется связь приложения с социальными сетями.

```
app.UseMicrosoftAccountAuthentication(
    clientId: "",
    clientSecret: "");

app.UseTwitterAuthentication(
   consumerKey: "",
   consumerSecret: "");

app.UseFacebookAuthentication(
   appId: "",
   appSecret: "");

app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
{
    ClientId = "",
    ClientSecret = ""
});
app.UseVkontakteAuthentication(new VkAuthenticationOptions
{
    // You should add "email" as a parameter for scope
    // if you are willing to receive user email
    Scope = new List<string>() { "email" },
    ClientId = "",
    ClientSecret = ""
});
```

Рисунок 2.4.3 – Конфигурация социальных сетей

После того, как авторизация реализована, необходимо настроить роли для пользователей. В базе данных в таблице AspNetRoles были добавлены необходимые нам роли:

— гость

— студент

— преподаватель

На рисунке 2.4.4 представлена таблица AspNetRoles.

	Id	Name
1	3	Guest
2	2	Student
3	1	Trainer

Рисунок 2.4.4 – Таблица AspNetRoles

Когда роли добавлены в базу, необходимо реализовать их присвоение пользователям. На рисунке 2.4.5 представлен листинг кода, демонстрирующий присвоение роли «гость» для пользователя.

```
if (result.Succeeded)
{
    result = await UserManager.AddLoginAsync(user.Id, info.Login);

    if (result.Succeeded)
    {
        result = await UserManager.AddToRoleAsync(user.Id, "Guest");
        if (result.Succeeded)
        {
            await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);
            return RedirectToLocal(returnUrl);
        }
    }
}
AddErrors(result);
```

Рисунок 2.4.5 – Присвоение пользователю роли «гость»

Далее можно приступить к реализации методов контроллера. Эта задача свелась к использованию нужных методов из наших сервисов. Это обеспечило простоту написания кода и значительно сократило скорость разработки приложения. А также, это поспособствовало уменьшению кода в методах контроллера.

Контроллерам необходимо присвоить атрибуты валидации, чтобы разрешить доступ к некоторым страницам только для определенных ролей. На рисунке 2.4.6 представлен код, показывающий метод с присвоенными ему атрибутами, включая атрибут валидации.

```
[HttpPost]
[ValidateAntiForgeryToken]
[Authorize(Roles = "Guest, Student, Trainer")]
0 references
public ActionResult Create(
    [Bind(Include = "Id,LastName,FirstName,MiddleName,University,Email,PhoneNumber,CourseId")] MitaJoinRequest
        mitaJoinRequest)
```

Рисунок 2.4.6 – Атрибуты валидации

Параллельно написанию контроллеров, также, производилась верстка страниц представления. ASP.NET MVC позволил генерировать строго типизированные представления. После генерации, страницы редактировались до нужного вида при помощи фреймворка Bootstrap, которые предоставляет уже готовые css-классы. Данный фреймоворк обеспечивает адаптивную верстку. Это значит, что страницы будут валидно отображаться как на персональных компьютерах, так и на мобильных устройствах.

ЗАКЛЮЧЕНИЕ

Композитная сервис-ориентированная архитектура – это подход к проектированию программных систем для организации уже существующих программных продуктов таким образом, чтобы разрозненные наборы сложных, распределенных систем и приложений можно было превратить в сеть интегрированных, простых и гибких ресурсов. Удачные SOA-решения объединяют информационные ресурсы, отвечая при этом в большей степени задачам бизнеса, что позволяет организациям выстраивать более тесное взаимодействие с заказчиками и поставщиками. В свою очередь это обеспечивает большую точность и доступность организации бизнес-процессов, позволяет принимать лучшие решения и помогает в обмене информацией между сотрудниками.

Практическое применение композитного подхода в разработке программного обеспечения является в настоящее время наиболее перспективным предметом изучения многих исследовательских организаций, а также поддерживается современными технологиями и паттернами разработки ПО (например, ASP.NET MVC, IoC-контейнеры, ORM и т.д.).

Использование описанного в выпускной квалификационной работе подхода и технологий в реальной сервис-ориентированной информационной системе MitaIS позволило достичь внушительных результатов в интеграции программных продуктов, значительно сократить сложность и стоимость разработки новых компонентов, а также поддержки «унаследованных» частей системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 СТО 4.2–07–2014 Система менеджмента качества. Общие требования к построению, изложению и оформлению документов учебной и научной деятельности. – Красноярск: ИПК СФУ, 2014. – 60 с.

2 Институт космических и информационных технологий СФУ [Электронный ресурс]. – Режим доступа: http://ikit.sfu-kras.ru/DPO.

3 Thomas, E. SOA: Principles of service design. / E. Thomas. – USA: Prentice Hall, 2007. – 608 с.

4 Northrop L. Software Product Lines Essentials. / L. Northrop. – Pittsburg: SEI Carnegie Mellon University, 2008. – 85 с.

5 Басс Л. Архитектура программного обеспечения на практике: 2-е издание. / Л. Басс, П. Клементс, Р. Кацман. – Санкт-Петербург: Питер, 2006. – 575 с.

6 Josuttis, N. SOA in practice. / N. Josuttis. – USA: O'Reilly, 2007. – 303 с.

7 Rotem, A. SOA patterns. / A. Rotem. – USA, 2012. – 296 с.

8 SOA Design patterns [Электронный ресурс]. – Режим доступа: http://soapatterns.org.

9 SOA Application Composite Architecture [Электронный ресурс]. – Режим доступа: http://docs.oracle.com/cd/E14571_01/integration.1111/e10223/arch_02.htm.

10 SOA in the Real World [Электронный ресурс]. – Режим доступа: http://www.microsoft.com/en-us/download/details.aspx?id=16187.

11 Lerman J. Programming Entity Framework. / J. Lerman. – USA: O'Reilly, 2010. – 914 с.

12 Фаулер, М. Архитектура корпоративных программных приложений. / М. Фаулер. – Москва: изд. дом «Вильямс», 2006. – 544 с.

13 Entity Framework [Электронный ресурс]. – Режим доступа: http://msdn.microsoft.com/en-US/data/ef.

14 Руководство Microsoft по проектироваию архитектуры приложений. – USA, 2010. 529 с.

15 Developer's Guide to Microsoft Enterprise Library. – USA, 2013. 232 с.

Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий

Кафедра «Информационные системы»

УТВЕРЖДАЮ Заведующий кафедрой ИС
_____С. А.Виденин
«__» _____2016 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
в форме бакалаврской работы

Студенту Васильеву Эдуарду Владимировичу

Группа: КИ12-13Б Направление: 09.03.02 «Информационные системы и технологии»

Тема выпускной квалификационной работы: «Разработка ИС для центра дополнительного и безотрывного образования по ИТ ИКИТ»

Утверждена приказом по университету № 4729/с от 05.04.2016.

Руководитель ВКР: С.А. Виденин, заведующий кафедрой «Информационные системы» ИКИТ СФУ.

Исходные данные для ВКР: список требований к разрабатываемой системе, методические указания научного руководителя.

Перечень разделов ВКР: введение, теоретическая часть, практическая часть**, заключение, список использованных источников.**

Перечень графического или иллюстрированного материала с указанием основных чертежей, плакатов, слайдов: презентация, выполненная в Microsoft Office PowerPoint 2013.

Руководитель ВКР С.А. Виденин

Задание принял к исполнению Э.В. Васильев

«___» _____2016 г.